Resource management is a general problem that exists at several levels in a real-time system.

- Shared resources internal to the runtime system:
  - CPU time
  - Memory pool (for dynamic allocation of memory)
  - Data structures (queues, tables, buffers, …)
  - I/O device access (ports, status registers, …)
- Shared resources specific to the application program:
  - Data structures (buffers, state variables, …)
  - Displays (to avoid garbled text if multiple tasks use it)
  - Entities in the application environment (seats in a cinema or an aircraft, a car parking facility, etc)

Resource management

Classification of resources:

- **Exclusive** access: there must be only one user at a time.
  - Examples: manipulation of data structures or I/O device registers
  - Exclusiveness is guaranteed through mutual exclusion
  - Program code that is executed while mutual exclusion applies is called a critical region
- **Shared** access: there can be multiple users at a time.
  - Examples: a car parking facility
  - The program code that makes sure that the number of users are within acceptable limits is itself a critical region

Operations for resource management:

- `acquire`: to request access to a resource
- `release`: to release a previously acquired resource

The `acquire` operation can be either blocking or non-blocking:

- Blocking: the task that calls `acquire` is blocked until the requested resource becomes available.
- Non-blocking: `acquire` returns a status code that indicates whether access to the resource was granted or not.

The `acquire` operation can be generalized so that the calling task can provide a priority. The task with the highest priority will then be granted access to the resource in case of simultaneous requests.
Problems with resource management:
- **Deadlock**: Tasks blocks each other and none of them can use the resource.
  - Deadlock can only occur if the tasks require access to more than one resource at the same time
  - Deadlock can be avoided by following certain guidelines
- **Starvation**: Some task is blocked because resources are always assigned to other (higher priority) tasks.
  - Starvation can occur in most resource management scenarios
  - Starvation can be avoided by granting access to resources in FIFO order

In general, deadlock and starvation are problems that must be solved by the program designer!

### Deadlock

**Conditions for deadlock to occur:**
1. **Mutual exclusion**
   - only one task at a time can use a resource
2. **Hold and wait**
   - there must be tasks that hold one resource at the same time as they request access to another resource
3. **No preemption**
   - a resource can only be released by the task holding it
4. **Circular wait**
   - there must exist a cyclic chain of tasks such that each task holds a resource that is requested by another task in the chain

**Guidelines for avoiding deadlock:**
1. Task should, if possible, only use one resource at a time.
2. If (1) is not possible, all tasks should request resources in the same order.
3. If (1) and (2) are not possible, special precautions should be taken to avoid deadlock. For example, resources could be requested using non-blocking calls.

The TinyTimber kernel can detect deadlock situations when a synchronous call is made. In such situations `SYNC()` will return a value of `-1`.

### Mutual exclusion

**Program constructs that provide mutual exclusion:**
- Ada 95 uses **protected objects**.
- Older languages (e.g. Modula-1, Concurrent Pascal) use **monitors**.
- Java uses **synchronized methods**, a simplified version of monitors.

When programming in languages (e.g. C and C++) that do not provide the constructs mentioned above, mechanisms provided by the real-time kernels or operating system must be used.
- POSIX offers **semaphores and methods with mutual exclusion**.
- The TinyTimber kernel offers **methods with mutual exclusion**.

To guarantee mutual exclusion in the implementation itself of the constructs mentioned above, machine-level methods must be used.
Protected objects:
- A protected object is a construct offered by Ada95.
- A protected object offers operations with mutual exclusion for data being shared by multiple tasks.
- A protected operation can be an entry, a procedure or a function. The latter is a read-only operation.
- Protected entries are guarded by a Boolean expression called a barrier.
- The barrier must evaluate to "true" to allow the entry body code to be executed. If the barrier evaluates to "false", the calling task will block until the barrier condition changes.

```
protected type Exclusive_Resource is
  entry Acquire;
  procedure Release;
private
  Busy : Boolean := false;
end Exclusive_Resource;
protected body Exclusive_Resource is
  entry Acquire when not Busy is
  begin
    Busy := true;
  end Acquire;
  procedure Release is
  begin
    Busy := false;
  end Release;
end Exclusive_Resource;
```

```
R : Exclusive_Resource;   -- instance of the resource
task A, B;    -- tasks using the resource
task body A is
begin
  R.Acquire;
  ...           -- critical region with code using the resource
  R.Release;
  ...
end A;
task body B is
begin
  R.Acquire;
  ...   -- critical region with code using the resource
  R.Release;
  ...
end B;
```

Monitors:
- A monitor is a construct offered by some (older) languages, e.g., Modula-1, Concurrent Pascal, Mesa.
- A monitor encapsulates data structures that are shared among multiple tasks and provides procedures to be called when a task needs to access the data structures.
- Execution of monitor procedures are done under mutual exclusion.
- Synchronization of tasks is done with a mechanism called condition variables.
Monitors vs. protected objects:
- Monitors are similar to protected objects in the sense that both are objects that can guarantee mutual exclusion during calls to procedures manipulating shared data.
- The difference between monitors and protected objects are in the way they handle synchronization:
  - Protected objects use entries with barriers (auto wake-up)
  - Monitors use condition variables (manual wake-up)
- Java offers a monitor-like construct:
  - Java’s synchronized methods correspond to monitor procedures
  - However, Java has no mechanism that corresponds to condition variables; a thread that gets woken up must check manually whether the resource is available.

Operations on condition variables:
- \texttt{wait(cond\_var)}: the calling task is blocked and is inserted into a FIFO queue corresponding to \texttt{cond\_var}.
- \texttt{send(cond\_var)}: wake up first task in the queue corresponding to \texttt{cond\_var}. No effect if the queue is empty.

Properties:
1. After a call to \texttt{wait} the monitor is released (e.g., other tasks may execute the monitor procedures).
2. A call to \texttt{send} must be the last statement in a monitor procedure.

Implementing an exclusive resource with a monitor:

```
monitor body Exclusive\_Resource is -- NOT Ada 95
  Busy : Boolean := false;
  CR : condition\_variable;
procedure Acquire is
  begin
    if Busy then
      Wait(CR);
    end if;
    Busy := true;
end Acquire;

procedure Release is
  begin
    Busy := false;
    Send(CR);
    end Release;
end Exclusive\_Resource;
```

```
... R : Exclusive\_Resource; -- instance of the resource

task A, B; -- tasks using the resource
task body A is
  begin
    A.Acquire;
    ... -- critical region with code using the resource
    B.Release;
    end A;
task body B is
  begin
    B.Acquire;
    ... -- critical region with code using the resource
    A.Release;
    end B;
...```
Semaphores:

- A semaphore is a passive synchronization primitive that is used for protecting shared and exclusive resources.
- Synchronization is done using two operations, `wait` and `signal`. These operations are atomic (indivisible) and are themselves critical regions with mutual exclusion.
- Semaphores are often used in run-time systems to implement more advanced mechanisms, e.g., protected objects or monitors.

Semaphores

A semaphore \( s \) is an integer variable with value domain \( \geq 0 \)

Atomic operations on semaphores:

- `Init(s,n)`: assign \( s \) an initial value \( n \)
- `Wait(s)`: if \( s > 0 \) then \( s := s - 1 \); else "block calling task"
- `Signal(s)`: if "any task that has called Wait(s) is blocked" then "allow one such task to execute"; else \( s := s + 1 \)

Implementing semaphores in Ada95:

```ada
protected type Semaphore (Initial : Natural := 0) is
  entry Wait;
  procedure Signal;
private
  Value : Natural := Initial;
end Semaphore;
protected body Semaphore is
  entry Wait when Value > 0 is
    begin
    Value := Value - 1;
    end Wait;
  procedure Signal is
    begin
    Value := Value + 1;
    end Signal;
  end Semaphore;
```

```ada
R : Semaphore(1);
task A, B;   -- tasks using the resource
  task body A is
    begin
      R.Wait;
      ...   -- critical region with code using the resource
      R.Signal;
      ...   -- critical region with code using the resource
    end A;
  task body B is
    begin
      R.Wait;
      ...   -- critical region with code using the resource
      R.Signal;
      ...   -- critical region with code using the resource
    end B;
```
Machine-level mutual exclusion

At some point all techniques mentioned so far need themselves be implemented using mutual exclusion. For this purpose there are two techniques are offered at the lowest (machine) level:

- **Disabling the processor’s interrupt service mechanism**
  - Should involve any interrupt that may lead to a task switch
  - Only works for single-processor systems

- **Atomic processor instructions**
  - For example: the test-and-set instruction
  - Variables can be tested and updated in one operation
  - Necessary for systems with two or more processors

Disabling processor interrupts

In single-processor systems, the mutual exclusion is guaranteed by disabling the processor’s interrupt service mechanism (“interrupt masking”) while the critical region is executed.

This way, unwanted task switches in the critical region (caused by e.g. timer interrupts) are avoided. However, all other tasks are unable to execute during this time.

Therefore, critical regions should only contain such instructions that really require mutual exclusion (e.g., code that handles the operations `wait` and `signal` for semaphores).

Note: this method is typically not used in multi-processor systems since interrupts are local to each processor.

Atomic processor instruction

In multi-processor systems with shared memory, a test-and-set instruction is used for handling critical regions.

A test-and-set instruction is a processor instruction that reads from and writes to a variable in one atomic operation.

The functionality of the test-and-set instruction can be illustrated by the following Ada procedure:

```ada
procedure testandset(lock, previous : in out Boolean) is
begin
  previous := lock;  -- lock is read and its value saved
  lock := true;  -- lock is set to "true"
  and testandset;
end
```

The combined read and write of `lock` must be atomic. In a multi-processor system, this is guaranteed by locking (disabling access to) the memory bus during the entire operation.
Atomic processor instruction

```haskell
lock : Boolean := false;  -- shared flag

task A, B;

task body A is
begin
  previous : Boolean;
  loop
    testandset(lock, previous);  -- A waits if critical region is busy
    exit when not previous;
  end loop;
  ...  -- critical region
    lock := false;  -- A leaves critical region
    ...  -- remaining program code
  end A;

task body B is
begin
  previous : Boolean;
  loop
    testandset(lock, previous);  -- A waits if critical region is busy
    exit when not previous;
  end loop;
  ...  -- critical region
    lock := false;  -- A leaves critical region
    ...  -- remaining program code
  end B;
```